**Event Loop**

Java script is a synchronous single threaded language it has one call stack, and it can do only one thing at a time.

Call stack waits for none, and it executes whatever comes it way, suppose if we want to execute a script/program after 5 sec in java script we do not have a timer for that

Diagram, text

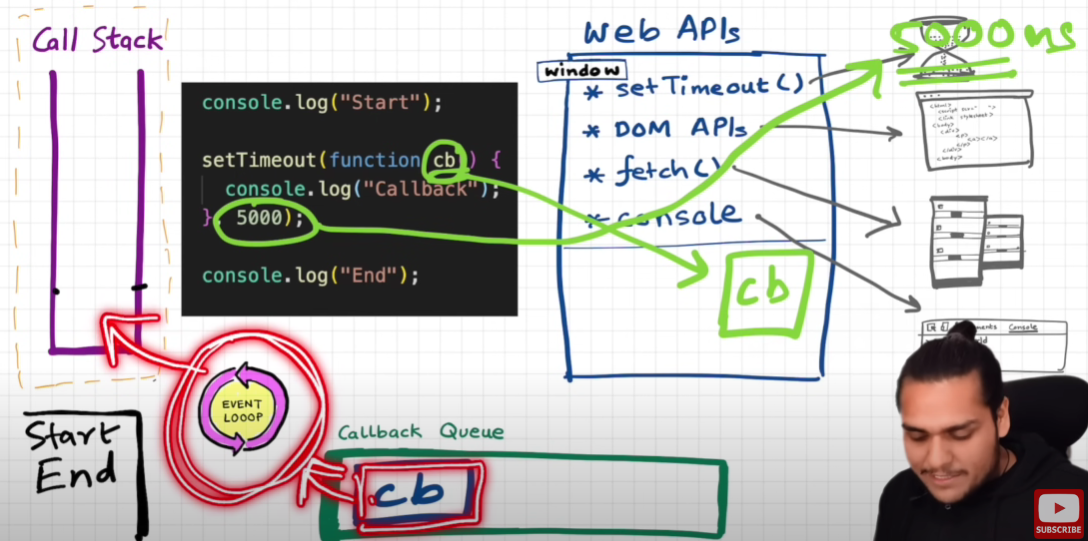
Description automatically generated

We can have access to timers with the help of web API’s. The browser has a java script inside it and call stack inside the js engine.

All this functionality of web API’s is attached to a object called window in java script.

So we can call window.setTimeout() to use timer functionality

If we are in global execution context we can directly call setTimeout().



Now when setTimeout() and call back function cb() is encountered with the help web api’s a timer and memory space to store callback function cb() is created.

Once the timer is finished by that time call stack is already empty and how is this cb() got loaded back into call stack?

For the purpose of this we have a Callback Queue and a Event Loop. Event Loop constantly searches inside the callback queue to find if anything is pending to be loaded on to the call stack.

function attachEventListners(){

    let count=0;

    document.getElementById("clickMe")

.addEventListener("click",function xyz(){

console.log("button", ++count);

});

}

Here document is a part of DOM API’S whenever it is encountered in code it goes into the html source code and picks the element by its ID.

Whenever call stack see’s this event listener it registers a call back in the memory and event “click” is attached to it.

If someone clicks on the button now, call back function gets loaded into the call back queue and event loop checks if call stack is empty and if yes it loads call back function into the stack.

**Now the question is why do we even need a callback queue? Why can’t event loop directly check with call back register and load it directly?**

We need it because if the button is clicked multiple times it has to loaded multiple times into the stack and in real time scenario there would be multiple event listeners from different programs we need to keep track of which has to be executed first.

fetch("https://api.netflix.com").then(function cbF(){

    console.log("CB Netflix");

});

Fetch() function requests a api call and returns a promise, once the promise is resolved, call back function will be executed.

One would expect that after the promise is resolved the function cbF() which is stored in callback register would be pulled into callback queue, but instead it would be pulled into microtask queue

Microtask queue is just like callback queue, but it has higher priority.

All the call back function which comes from promises go inside the microtask queue.

Mutation observer constantly checks if there is some mutation in the dom tree or not. If there is some mutation in the dom tree.

So, the callback functions from the promises and mutation observer gets into microtask queue.

Starvation: If the microtask callback creates another microtask callback inside it and then one more inside it. If this loop goes on the function in callback queue never gets a chance and it is starving.